

Network programming (IT423+IT432)

Spring 2017

Dr. Islam Taj-Eddin

IT Dept., FCI, Assiut Univ.

Streams

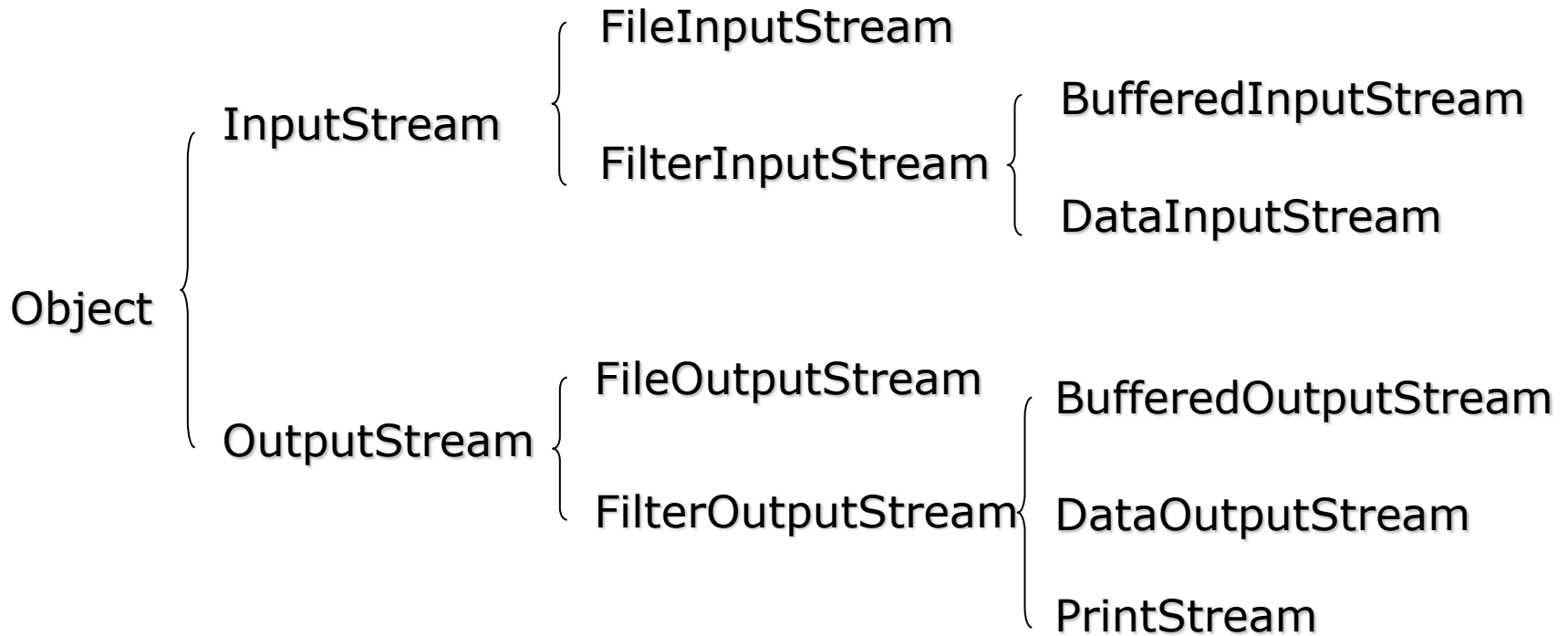
I/O and streams

- ▶ Large part of network programming is concerned with data movement from one system to another.
- ▶ I/O in Java is built on streams;
 - input streams read data
 - output streams write data.
- ▶ Java has different classes to deal with input and output streams.

Java I/O Streams

- I/O Streams
 - Byte stream: Input Stream and Output Stream
 - Filter Stream
 - Buffered Stream
 - Data Stream
 - Print Stream
 - File Stream
 - Character Stream: Reader and Writer
 - Input Stream Reader and Output Stream Writer
 - Buffered Reader/Writer
 - File Reader/Writer

Byte Streams



InputStream Class

- `java.io.InputStream` is an abstract class for all input streams.
- It contains methods for reading in raw bytes of data from input stream: key board, files, network client.
 - `public abstract int read()`
 - `public int read (byte[] buf)`
 - `public int read(byte[] buf, int offset, int length)`

InputStream Class

- public long skip(long n)
 - . skip n number of bytes
- public int available()
 - . how many bytes can be read before blocking
- public void close()
- public synchronized void mark (int readlimit)
 - . bookmark current position in the stream
- public boolean markSupported()
- public synchronized void reset()
 - . rewind the stream to the marked position
- All but the last two methods throw an IOException.

The read() method

- The basic read() method reads a single unsigned byte of data and returns the integer value of the unsigned byte.
- This is a number between 0 and 255
- Returns a -1 if the end of a stream is encountered.
- The method blocks until input data are available, the end of stream is detected or an exception is thrown.

The read() method

```
int[] data = new int[10];  
for (int i =0; i <data.length, i++)  
    data [i]= System.in.read( );  
}
```

BufferedInputStream



- This code reads in 10 bytes from the System.in input stream and stores it in the int array data.
- Notice that although read() reads in a byte, it returns a value of type int. If you want the raw byte, cast the int into a byte.

The read() method

- read() has a possibility of throwing an exception.

```
try {  
    int data[] = new int[10] ;  
    for (int i=0; i<data.length; i++) {  
        int datum = System.in.read();  
        if (datum == -1) break;  
        data[i] = datum;  
    }//for  
}//try  
catch (IOException e) {  
    System.err.println(e);  
}
```



End of stream

The read() method

- The value of -1 is returned when the end of stream is reached. This can be used as a check for the stream end.
- Remember that read() blocks. So if there is any other important work to do in your program, try to put your I/O in a separate thread.
- read() is abstract method defined in InputStream. This means you can't instantiate InputStream directly: work with one of its subclasses instead.

Echo Example(I)


```
import java.io.*,
```

```
public class Echo {  
    public static void main(String[] args){  
        echo(System.in);  
    }//main
```

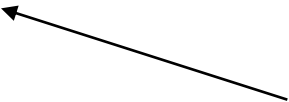
```
public static void echo(InputStream is) {  
    try {
```

```
        for (int j = 0; j < 20; j++) {int i = is.read( );
```

BufferedInputStream



An instance of a subclass
of InputStream
(remember: upcasting)



Echo Example(2)

```
// -1 returned for end of stream
if (i == -1)
    break;
char c = (char) i ;
System.out.print(c);
} //for loop
} //try
catch (IOException e){
    System.err.println();
} //catch
System.out.println( );
} //echo method
} //Echo class
```

Reading Multiple Bytes

- Since accessing I/O is slow in comparison to memory access, limiting the number of reads and writes is essential.
- The basic `read()` method only reads in a byte at a time.
- The following two overloading `read()` methods read in multiple bytes into an array of bytes.
 - `public int read(byte b[])`
 - `public int read(byte b[], int offset, int length)`

Reading Multiple Bytes

- The first method tries to read enough bytes to fill the array `b[]`.

```
try {  
    byte[ ] b = new byte[10];  
    int j = System.in.read(b);  
}  
catch (IOException e){ }
```

- This method blocks until data are available just like the `read()` method.

Reading Multiple Bytes

- The second method reads `length` bytes from the input stream and stores them in the array `b[]` starting at the location `offset`.

```
try { //what does this loop do
    byte[] b = new byte[100];
    int offset = 0;
    while (offset < b.length) {
        int bytesRead = System.in.read(b, offset, b.length - offset);
        if (bytesRead == -1) break;
        offset += bytesRead; } //while
    catch (IOException e) { }
```

Closing Input Streams

- For well behaved programs, all **streams** should be **closed** before **exiting** the program.
- **Allows OS to free any resources associated with the stream.**
- Use the **close()** method
 - `public void close()` throws `IOException`
- Not all streams have to be closed.
 - **System.in** does not have to be closed.

Closing Input Streams

```
try {  
    URL u = new URL("http://java.sun.com");  
    InputStream in = u.openStream();  
    / read from stream ...  
    in.close();  
}  
catch (IOException e){}
```

- Once an input stream has been closed, you can no longer read from it. Doing so will cause an `IOException` to be thrown.

Reading from File Input Streams

```
import java.io.*;
class FileInputStreamDemo {
    public static void main(String args[]) {
        try { //Create a file input stream
            FileInputStream fis = new FileInputStream(args[0]);
            //read 12 byte from the file
            int i;
            while ((i = fis.read()) != -1)
                {System.out.println(i);}
            //Close file output stream
            fis.close();
        } catch (Exception e) {System.out.println("Exception: " + e);}
    }
}
```

Reading from Buffered Input Streams

```
import java.io.*;
class FileBufferedStreamDemo {
    public static void main(String args[]) {
        try { //Create a file input stream
            FileInputStream fis = new FileInputStream(args[0]);
            //Create a buffered input stream
            BufferedInputStream bis = new BufferedInputStream(fis);
            //read 12 byte from the file
            int i;
            while ((i = bis.read()) != -1)
                {System.out.println(i);}
            //Close file output stream
            fis.close();
        } catch (Exception e) {System.out.println("Exception: " + e);}
    }
}
```

Reading from Data Input Streams

```
import java.io.*;
class DataInputStreamDemo {
    public static void main(String args[]) {
        try { //Create a file input stream
            FileInputStream fis = new FileInputStream(args[0]);
            //Create a data input stream
            DataInputStream dis = new DataInputStream(fis);
            //read and display data
            System.out.println(dis.readBoolean());
            System.out.println(dis.readByte());
        }
    }
}
```

Reading from Data Input Streams

```
System.out.println(dis.readChar());  
System.out.println(dis.readDouble());  
System.out.println(dis.readFloat());  
System.out.println(dis.readInt());  
System.out.println(dis.readLong());  
System.out.println(dis.readShort());  
//Close file input stream  
fis.close();
```

```
}catch(Exception e) {System.out.println("Exception: " + e);}}
```

```
}}
```

Output Streams

- `java.io.OutputStream` class sends raw bytes of data to a target such as the console, a file, or a network server.
- Methods within this class are:
 - `public abstract void write(int b)`
 - `public void write(byte b[])`
 - `public void write(byte b[], int offset, int length)`
 - `public void flush()`
 - `public void close()`
- All methods throw an `IOException`

Output Streams

- The `write()` methods sends **raw bytes of data** to whomever is listening to the stream.
- Sometimes for performance reasons, the operating system **buffers** output streams.
- When the buffer fills up, the data are all written at once.
- The `flush()` method will force the data to be written whether the buffer is full or not.

Writing to Output Streams

- The fundamental method in `OutputStream` is `write()`
- `public abstract void write(byte b)`
- This method writes a single unsigned byte of data that should be between 0 and 255.
- Larger numbers are reduced modulo 256 before writing.

Ascii Chart Example

```
import java.io.*;
public class AsciiChart{
    public static void main(String args[]) {
        for (int i=32; i<127; i++)
            System.out.write(i);
            //break line after every 8 characters
        if (i%8 == 7) System.out.write('\n');
        else System.out.write('\t');
        }//for
        System.out.write('\n');
    }//main
}//class
```

Writing Arrays of Bytes

- The two remaining write methods write multiple bytes of data.
 - Public void write(byte b[])
 - Public void write(byte b[], int offset, int length)
- The first writes an entire byte array of data, while the second writes a sub-array of data starting at offset and continuing for length bytes.
- Remember that these methods write bytes, so data must be converted into bytes.

AsciiArray Example

```
import java.io.*;
public class AsciiArray{
    public static void main(String args[]) {
        int index=0;
        byte[] b = new byte[(127-31)*2];
        for (int i=32; i<127; i++) {
            b[index++] = (byte)i;
            //break line after every 8 characters
            if (i%8==7) b[index++] = (byte)'\n';
            else b[index++] = (byte) '\t';
        }
    }
}
```

AsciiArray Example

```
    }//for
    b[index++] = (byte) '\n';
    try {
        System.out.write(b);
    }
    catch(IOException e) {}
} //main
} //class
```

- The output is the same as AsciiChart.

Writing to File Output Streams

```
import java.io.*;
class FileOutputStreamDemo {
    public static void main(String args[]) {
        try { //Create a file output stream
            FileOutputStream fos = new FileOutputStream(args[0]);
            //Write 12 byte to the file
            for (int i = 0; i < 12; i++) {
                fos.write(i);}
            //Close file output stream
            fos.close();
        } catch (Exception e) {System.out.println("Exception: " + e);}
    }
}
```

Flushing and Closing Output Streams

- As mentioned, many operating systems buffer output data to improve performance.
- Rather than sending a bytes at a time, bytes are accumulated until the buffer is full, and one write occurs.
- The flush() method forces the data to be written even if the buffer is not full.
 - public void flush() throws IOException
- Like input streams, output streams should be closed. For output streams, closing them will also flush the contents of the buffer.

Filter Streams

- `java.io.FilterInputStream` and `java.io.FilterOutputStream` are subclasses of `InputStream` and `OutputStream`, respectively.
- These classes are rarely used, but their subclasses are extremely important.

Filter Streams Classes

- **Buffered Streams**
 - These classes will buffer reads and writes by first reading the data into a buffer (array of bytes)
- **Data Streams**
 - These classes read and write primitive data types and Strings.
- **Print Stream**
 - referenced by `System.out` and `System.err`.
 - It uses the platform's default character encoding to convert characters into bytes.

Buffered Streams

- Buffered input stream read more data than initially needed and store them in a buffer.
- So when the buffered stream's `read()` method is called, the data is removed from the buffer rather than from the underlying system.
- When the buffer is empty, the buffered stream refills the buffer.
- Buffered output stream store data in an internal byte array until the buffer is full or the stream is flushed. The data is then written out once.

Buffered Streams

- Constructors
 - `BufferedInputStream(InputStream in)`
 - `BufferedInputStream(InputStream in, int size)`
 - `BufferedOutputStream(OutputStream out)`
 - `BufferedOutputStream(OutputStream out, int size)`
- The size argument is the size of the buffer.
- If not specified, a default of 512 bytes is used.

Buffered Streams

- Example:

```
URL u=new URL("http://java.sun.Com");  
BufferedInputStream bis;  
bis= new BufferedInputStream(u.openStream( ), 256)
```

- BufferedInputStream and BufferedOutputStream do not declare any new methods but rather override methods from InputStream and OutputStream, respectively.

Writing to Buffered Output Streams

```
import java.io.*;
class BufferedOutputStreamDemo {
    public static void main(String args[]) {
        try { //Create a file output stream
            FileOutputStream fos = new FileOutputStream(args[0]);
            //Create a buffered output stream
            BufferedOutputStream bos = new BufferedOutputStream(fos);
            //Write 12 byte to the file
            for (int i = 0; i < 12; i++) {
                bos.write(i);}
            //Close file output stream
            bos.close(); fos.close();
        } catch (Exception e) {System.out.println("Exception: " + e);}
    }
}
```

Data Streams

- `java.io.DataInputStream` and `java.io.DataOutputStream` read and write primitive data types and strings using the `java.io.DataInput` and `java.io.DataOutput` interfaces, respectively.

Data Streams

- Generally you use `DataInputStream` to read data written by `DataOutputStream`
- `public DataInputStrem(InputStream in)`
- `public DataOutputStream(OutputStream out)`
- The usual methods associated with input and output streams are present in data stream as well.
- However, data streams have other methods that allow them to read and write primitive type.

Writing to Data Output Streams

```
import java.io.*;
class DataOutputStreamDemo {
    public static void main(String args[]) {
        try { //Create a file output stream
            FileOutputStream fos = new FileOutputStream(args[0]);
            //Create a data output stream
            DataOutputStream dos = new DataOutputStream(fos);
            //Write various types of data to the file
            dos.writeBoolean(false);
            dos.writeByte(Byte.MAX_VALUE);
        }
    }
}
```

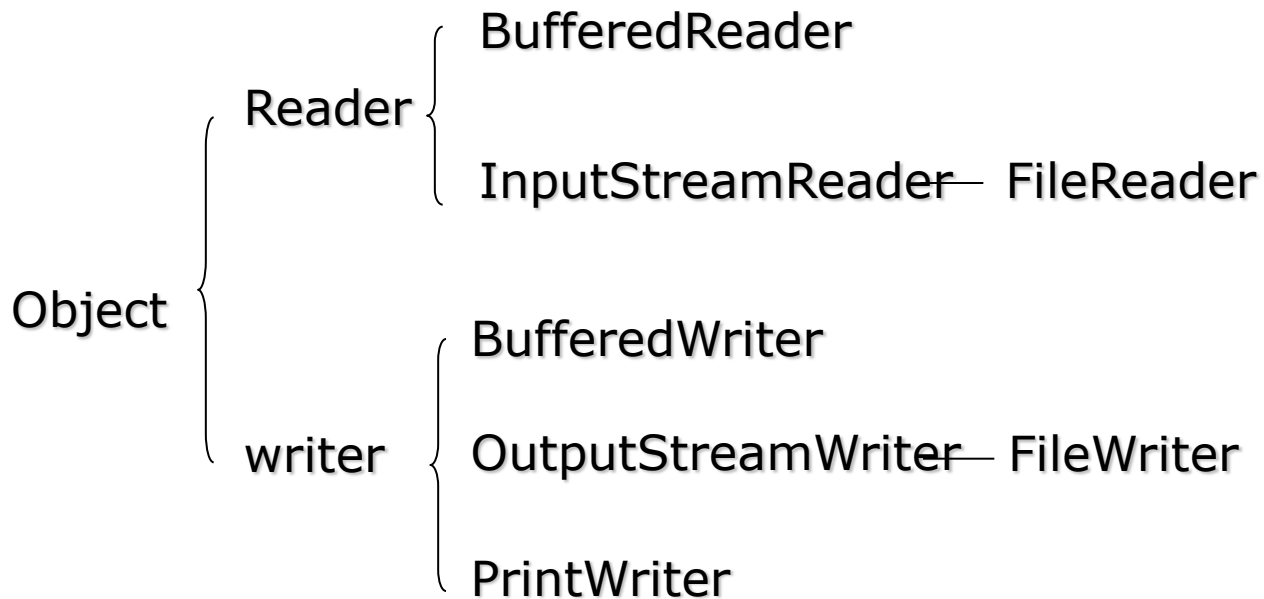
Writing to Data Output Streams

```
dos.writeChar('A');
dos.writeDouble(Double.MAX_VALUE);
dos.writeFloat(Float.MAX_VALUE);
dos.writeInt(int.MAX_VALUE);
dos.writeLong(Long.MAX_VALUE);
dos.writeShort(Short.MAX_VALUE);
//Close file output stream
fos.close();
} catch (Exception e) {System.out.println("Exception: " + e);}
}}
```


Print Streams

- Allows very simple printing of both primitive values, objects, string literals.
- There are many overloaded `print()` and `println()` methods.
- This method is deprecated in Java 1.1.
- The biggest problem with this class is that it does not properly handle international character sets.
- Use the `PrintWriter` class instead.

Character Streams



Readers and Writers

- Classes that read and write character based data.
- These characters can have varying widths depending on the character set being used.
- Readers and writers know how to handle many different character sets.

Reader Class

- `java.io.Reader`
- This class is deliberately similar to the `java.io.InputStream` class.
- Methods in the Reader class are similar to the `InputStream` class except that the methods work on characters not bytes.

Writer Class

- `Java.io.Writer`
- This class is similar to the `java.io.OutputStream` class.
- Methods in the `Writer` class now work on characters and not bytes.

InputStreamReader

- `java.io.InputStreamReader` acts as a translator between byte streams and character streams.
- It reads bytes from the input stream and translates them into characters according to a specified character encoding.

InputStreamReader Class

- You can set the encoding scheme or you can use the platforms default setting.
- `public InputStreamReader(InputStream in)`
- `public InputStreamReader(InputStream in, String enc)`
throws `UnsupportedEncodingException`

OutputStreamWriter

- `java.io.OutputStreamWriter` will write bytes of data to the output stream after translating the characters according to the specified encoding.
- `public OutputStreamWriter(OutputStream out)`
- `public OutputStreamWriter(OutputStream out, String enc)`
throws `UnsupportedEncodingException`

Buffered Reads/Writes

- There are classes that allow for a more efficient reading and writing of characters by buffering.
- `java.io.BufferedReader`
- `java.io.BufferedWriter`
- These classes are similar to the Buffered Stream classes.
- Most notable for the `readLine()` Method. This allows data to be read a line at a time.
- `public String readLine() throws IOException`

Buffered Reads/Writes

```
import java.io.*;

public class StringInputFile {
    public static void main(String[] arg) throws Exception {
        PrintStream backup;
        FileOutputStream backupFileStream;
        File backupFile;
        backupFile = new File("backup");
        backupFileStream = new FileOutputStream(backupFile);
        backup = new PrintStream(backupFileStream);
```

Buffered Reads/Writes

```
System.out.println("This is my first data file");
    backup.println("This is my first data file");
    System.out.println("... but it won't be my last");
    backup.println("... but it won't be my last");
}
}
```

Buffered Reads/Writes

Writing output to a file involves three steps as follows:

- Create a File object
- Create a FileOutputStream object
- Create a PrintStream object

Buffered Reads/Writes

```
import java.io.*;

public class StringInputFile {
    public static void main(String[] arg) throws Exception {
        InputStreamReader backup;
        BufferedReader br;
        FileInputStream backupFileStream;
        File backupFile;

        String inputline;
```

Buffered Reads/Writes

```
backupFile = new File("backup");  
backupFileStream = new FileInputStream(backupFile);  
backup = new InputStreamReader(backupFileStream);  
br = new BufferedReader(backup);  
inputline = br.readLine();  
System.out.println(inputline);  
inputline = br.readLine();  
System.out.println(inputline);  
}
```

```
}
```

Buffered Reads/Writes

Reading data from a file involves three steps as follows:

- Create a `FileInputStream` or `BufferedInputStream` object
- Create a `InputStreamReader` object which we use to
- Create a `BufferedReader` object

Example: Send Data(I)

```
import java.net.*; import java.io.*;

public class SendData extends Thread {
    Socket sock;

    public SendData (Socket sock) {

        this.sock = sock;
    } //SendData constructor

    public void run() {
        string line;
```


Example: Send Data(2)

```
try {  
    OutputStreamWriter outw=new  
    outputStreamwriter(sock.getOutputStream());  
    BufferedWriter sockout=new  
    BufferedWriter(outw);  
    InputStreamReader inr = new InputStreamReader(System.in);  
    BufferedReader in = new BufferedReader(inr);  
  
    while ((line = in.readLine()) != null) {  
  
        sockout.write(line+ "\n");
```

Example: Send Data(3)

```
        socket.flush(); yield( );
    } //while
} //try
catch (java.io.IOException e) {
    System.out.println(e);
    System.exit(0);
} //catch
} //run
} //SendData
```

Example: Receive Data(I)

```
import java.net.*;  
import java.io.*;
```

```
public class RcveData extends Thread {  
    Socket sock;  
  
    public RcveData(Socket sock) {  
        this.sock = sock;  
    }  
}
```

```
public void run() {  
  
    String line;
```

Example: Receive Data(2)

```
try {  
    InputStreamReader inr = new  
    InputStreamReader(sock.getInputStream());  
  
    BufferedReader in = new BufferedReader(inr);  
  
    while ((line = in.readLine()) != null) {  
  
        System.out.print(mReceiving:  
        System.out.println(line);  
        yield();  
    }//while  
}//try
```

Example: Receive Data(3)

```
catch (java.io.IOException e) {
```

```
    System.out.println(e);
```

```
    System.exit(0);
```

```
I        }//catch
```

```
        }//run
```

```
}//RCVeData
```